ARIANN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing

Théo Ryffel, David Pointcheval, Francis Bach

ARIANN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing

Théo Ryffel[1, 2], David Pointcheval[2, 1] and Francis Bach[1, 2]

[1]INRIA, Paris, France
[2]Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France
{theo.ryffel,david.pointcheval,francis.bach}@ens.fr

Abstract

We propose ARIANN, a low-interaction framework to perform private training and inference of standard deep neural networks on sensitive data. This framework implements semi-honest 2-party computation and leverages function secret sharing, a recent cryptographic protocol that only uses lightweight primitives to achieve an efficient online phase with a single message of the size of the inputs, for operations like comparison and multiplication which are building blocks of neural networks. Built on top of PyTorch, it offers a wide range of functions including ReLU, MaxPool and BatchNorm, and allows to use models like AlexNet or ResNet18. We report experimental results for inference and training over distant servers. Last, we propose an extension to support n-party private federated learning.

1 Introduction

The massive improvements of cryptography techniques for secure computation over sensitive data [15, 13, 28] have spurred the development of the field of privacy-preserving machine learning [45, 1]. Privacy-preserving techniques have become practical for concrete use cases, thus encouraging public authorities to use them to protect citizens' data, for example in covid-19 apps [27, 17, 38, 39].

However, tools are lacking to provide end-to-end solutions for institutions that have little expertise in cryptography while facing critical data privacy challenges. A striking example is hospitals which handle large amounts of data while having relatively constrained technical teams. Secure multi-party computation (SMPC) is a promising technique that can efficiently be integrated into machine learning workflows to ensure data and model privacy, while allowing multiple parties or institutions to participate in a joint project. In particular, SMPC provides intrinsic shared governance: because data are shared, none of the parties can decide alone to reconstruct it. This is particularly suited for collaborations between institutions willing to share ownership on a trained model.

Use case. The main use case driving our work is the collaboration between healthcare institutions such as hospitals or clinical research laboratories. Such collaboration involves a model owner and possibly several data owners like hospitals. As the model can be a sensitive asset (in terms of intellectual property, strategic asset or regulatory and privacy issues), standard federated learning [29, 7] that does not protect against model theft or model retro-engineering [24, 18] is not suitable.

Based on this use case, we will make the following assumptions: first, the parties involved in the computation are in the *Wide Area Network (WAN) setting*: they are located typically in different countries, are well connected and can communicate large amounts of information over the network with a reasonable latency. They also have access to relatively limited computing power compared

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

to data centers, but are likely to remain online for long periods of time. Last, parties are *honest-but-curious*, [20, Chapter 7.2.2] and care about their reputation. Hence, they have little incentive to deviate from the original protocol, but they will use any information available in their own interest.

Contributions. By leveraging function secret sharing (FSS) [9, 10], we propose the first low-interaction framework for private deep learning which drastically reduces communication to a single round for basic machine learning operations, and achieves the first private evaluation benchmark on ResNet18.

- We build on existing work on function secret sharing to design compact and efficient algorithms for comparison and multiplication, which are building blocks of neural networks. They are highly modular and can be assembled to build complex workflows.
- We show how these blocks can be used in machine learning to implement operations for secure evaluation and training of arbitrary models on private data, including MaxPool and BatchNorm. *We achieve single round communication for comparison, convolutional or linear layers.*
- Last, we provide an implementation[1] and demonstrate its practicality both in LAN (local area network) and WAN settings by running secure training and inference on CIFAR-10 and Tiny Imagenet with models such as AlexNet [31] and ResNet18 [22].

Related work. Related work in privacy-preserving machine learning encompasses SMPC and homomorphic encryption (HE) techniques.

HE only needs a single round of interaction but does not support efficiently non-linearities. For example, nGraph-HE [5] and its extensions [4] build on the SEAL library [44] and provide a framework for secure evaluation that greatly improves on the CryptoNet seminal work [19], but it resorts to polynomials (like the square) for activation functions.

SMPC frameworks usually provide faster implementations using lightweight cryptography. MiniONN and DeepSecure [34, 41] use optimized garbled circuits [50] that allow very few communication rounds, but they do not support training and alter the neural network structure to speed up execution. Other frameworks such as ShareMind [6], SecureML [36], SecureNN [47] or more recently FALCON [48] rely on additive secret sharing and allow secure model evaluation and training. They use simpler and more efficient primitives, but require a large number of rounds of communication, such as 11 in [47] or $5 + \log_2(l)$ in [48] (typically 10 with $l = 32$) for ReLU. ABY [16], Chameleon [40] and more recently ABY^3 [35] mix garbled circuits, additive or binary secret sharing based on what is most efficient for the operations considered. However, conversion between those can be expensive and they do not support training except ABY^3. Last, works like Gazelle [26] combine HE and SMPC to make the most of both, but conversion can also be costly.

Works on trusted execution environment are left out of the scope of this article as they require access to dedicated hardware [25]. Data owners which cannot afford these secure enclaves might be reluctant to use a cloud service and to send their data.

2 Background

Notations. All values are encoded on n bits and live in $\mathbb{Z}_{2^n}$. The notation $[\![x]\!]$ denotes 2-party additive secret sharing of x, i.e., $[\![x]\!] = ([\![x]\!]_0, [\![x]\!]_1)$ where the shares $[\![x]\!]_j$ are random in $\mathbb{Z}_{2^n}$, are held by distinct parties and verify $x = [\![x]\!]_0 + [\![x]\!]_1 \bmod 2^n$. In return, $x[i]$ refers to the i-th bit of x.

Function secret sharing. Unlike classical data secret sharing, where a shared input $[\![x]\!]$ is applied on a public f, function secret sharing applies a public input x on a private shared function $[\![f]\!]$. Shares or *keys* $([\![f]\!]_0, [\![f]\!]_1)$ of a function f satisfy $f(x) = [\![f]\!]_0(x) + [\![f]\!]_1(x) \bmod 2^n$. Both approaches output a secret shared result.

Let us take an example: say Alice and Bob respectively have shares $[\![y]\!]_0$ and $[\![y]\!]_1$ of a private input y, and they want to compute $[\![y \geq 0]\!]$. While $y \in \mathbb{Z}_{2^n}$, we can associate it to an n-bit signed integer in $[-2^{n-1}, 2^{n-1} - 1]$, where the most significant bit (MSB) is a sign bit. They first mask their shares using a random mask $[\![\alpha]\!]$, by computing $[\![y]\!]_0 + [\![\alpha]\!]_0$ and $[\![y]\!]_1 + [\![\alpha]\!]_1$, and then reveal these values to reconstruct $x = y + \alpha$. Next, they apply this public x on their function shares $[\![f_\alpha]\!]_j$ of $f_\alpha : x \to (x \geq \alpha)$, to obtain a shared output $([\![f_\alpha]\!]_0(x), [\![f_\alpha]\!]_1(x)) = [\![f_\alpha(y+\alpha)]\!] = [\![(y+\alpha) \geq \alpha]\!] = [\![y \geq 0]\!]$.

[1] All code and implementations will be made available on GitHub.

[9, 10] have shown the existence of such function shares for comparison which perfectly hide y and the result. From on now, to be consistent with the existing literature, we will denote the function keys $(\mathsf{k}_0, \mathsf{k}_1) := (\llbracket f \rrbracket_0, \llbracket f \rrbracket_1)$.

Note that for a perfect comparison, $y + \alpha$ should not wrap around and become negative. Because y is in practice small compared to the n-bit encoding amplitude, the failure rate is less than one comparison in a million, as detailed in Appendix C.1.

Security model. We consider security against *honest-but-curious* adversaries, i.e., parties following the protocol but trying to infer as much information as possible about others' input or function share. This is a standard security model in many SMPC frameworks [6, 3, 40, 47] and is aligned with our main use case: parties that would not follow the protocol would face major backlash for their reputation if they got caught. The security of our protocols relies on indistinguishability of the function shares, which informally means that the shares received by each party are computationally indistinguishable from random strings. A formal definition of the security is given in [10].

About malicious adversaries, i.e., parties who would not follow the protocol, as all the data available are random, they cannot get any information about the inputs of the other parties, including the parameters of the evaluated functions, unless the parties reconstruct some shared values. The later and the fewer values are reconstructed, the better it is. As mentioned by [11], our protocols could be extended to guarantee *security with abort* against malicious adversaries using MAC authentication [15], which means that the protocol would abort if parties deviated from it.

3 Function Secret Sharing Primitives

Our algorithms for private equality and comparison are built on top of the work of [10], so the security assumptions are the same as in this article. However, our protocols achieve higher efficiency by specializing on the operations needed for neural network evaluation or training.

3.1 Equality test

We start by describing private equality which is slightly simpler and gives useful hints about how comparison works. The equality test consists in comparing a public input x to a private value α. Evaluating the input using the function keys can be viewed as walking a binary tree of depth n, where n is the number of bits of the input (typically 32). Among all the possible paths, the path from the root down to α is called the *special path*. Figure 1 illustrates this tree and provides a compact representation which is used by our protocol, where we do not detail branches for which all leaves are 0. Evaluation goes as follows: two evaluators are each given a function key which includes a distinct initial random state $(s, t) \in \{0,1\}^\lambda \times \{0,1\}$. Each evaluator starts from the root, at each step i goes down one node in the tree and updates his state depending on the bit $x[i]$ using a common *correction word* $CW^{(i)} \in \{0,1\}^{2(\lambda+1)}$ from the function key. At the end of the computation, each evaluation outputs t. As long as $x[i] = \alpha[i]$, the evaluators stay on the special path and because the input x is public and common, they both follow the same path. If a bit $x[i] \neq \alpha[i]$ is met, they leave the special path and should output 0 ; else, they stay on it all the way down, which means that $x = \alpha$ and they should output 1.

The main idea is that while they are on the special path, evaluators should have states (s_0, t_0) and (s_1, t_1) respectively, such that s_0 and s_1 are i.i.d. and $t_0 \oplus t_1 = 1$. When they leave it, the correction word should act to have $s_0 = s_1$ but still indistinguishable from random and $t_0 = t_1$, which ensures $t_0 \oplus t_1 = 0$. Each evaluator should output its t_j and the result will be given by $t_0 \oplus t_1$. The formal description of the protocol is given below and is composed of two parts: first, in Algorithm 1, the KeyGen algorithm consists of a preprocessing step to generate the functions keys, and then, in Algorithm 2, Eval is run by two evaluators to perform the equality test. It takes as input the private share held by each evaluator and the function key that they have received. They use $G : \{0,1\}^\lambda \to \{0,1\}^{2(\lambda+1)}$, a pseudorandom generator, where the output set is $\{0,1\}^{\lambda+1} \times \{0,1\}^{\lambda+1}$, and operations modulo 2^n implicitly convert back and forth n-bit strings into integers.

Intuitively, the correction words $CW^{(i)}$ are built from the expected state of each evaluator on the special path, i.e., the state that each should have at each node i if it is on the special path given some initial state. During evaluation, a correction word is applied by an evaluator only when it has $t = 1$. Hence, on the special path, the correction is applied only by one evaluator at each bit.

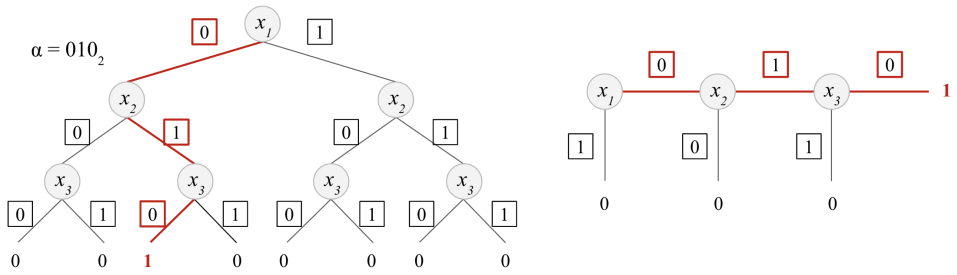

Figure 1: *(left)* Binary decision tree with the special path for $n = 3$. Given an input $x = (x_1...x_n)_2$, at each level i, one should take the path labeled by the value in the square equal to the bit value x_i. *(right)* Flat representation of the tree.

Initialisation: Sample random $\alpha \xleftarrow{\$} \mathbb{Z}_{2^n}$

Sample random $s_j^{(1)} \xleftarrow{\$} \{0,1\}^\lambda$ and set $t_j^{(1)} \leftarrow j$, for $j = 0, 1$

for $i = 1..n$ **do**

  $(s_j^{\mathsf{L}} \,||\, t_j^{\mathsf{L}}, s_j^{\mathsf{R}} \,||\, t_j^{\mathsf{R}}) \leftarrow G(s_j^{(i)}) \in \{0,1\}^{\lambda+1} \times \{0,1\}^{\lambda+1}$, for $j = 0, 1$

  if $\alpha[i]$ **then** $cw^{(i)} \leftarrow (0^\lambda \,||\, 0, s_0^{\mathsf{L}} \oplus s_1^{\mathsf{L}} \,||\, 1)$

  else $cw^{(i)} \leftarrow (s_0^{\mathsf{R}} \oplus s_1^{\mathsf{R}} \,||\, 1, 0^\lambda \,||\, 0)$;

  $CW^{(i)} \leftarrow cw^{(i)} \oplus G(s_0^{(i)}) \oplus G(s_1^{(i)}) \in \{0,1\}^{\lambda+1} \times \{0,1\}^{\lambda+1}$

  $state_j \leftarrow G(s_j^{(i)}) \oplus (t_j^{(i)} \cdot CW^{(i)}) = (state_{j,0}, state_{j,1})$, for $j = 0, 1$

  Parse $s_j^{(i+1)} \,||\, t_j^{(i+1)} = state_{j,\alpha[i]} \in \{0,1\}^{\lambda+1}$, for $j = 0, 1$

$CW^{(n+1)} \leftarrow (-1)^{t_1^{(n+1)}} \cdot \left(1 - s_0^{(n+1)} + s_1^{(n+1)}\right) \bmod 2^n$

return $\mathsf{k}_j \leftarrow [\![\alpha]\!]_j \,||\, s_j^{(1)} \,||\, CW^{(1)} \,||\, \cdots \,||\, CW^{(n+1)}$, for $j = 0, 1$

Algorithm 1: KeyGen: key generation for equality to α

If at step i, the evaluator stays on the special path, the correction word compensates the current states of both evaluators by xor-ing them with themselves and re-introduces a pseudorandom value s (either $s_0^{\mathsf{R}} \oplus s_1^{\mathsf{R}}$ or $s_0^{\mathsf{L}} \oplus s_1^{\mathsf{L}}$), which means the xor of their states is now $(s, 1)$ but those states are still indistinguishable from random. On the other hand, if $x[i] \neq \alpha[i]$, the new state takes the other half of the correction word, so that the xor of the two evaluators states is (0, 0). From there, they have the same states and both have either $t = 0$ or $t = 1$. They will continue to apply the same corrections at each step and their states will remain the same with $t_0 \oplus t_1 = 0$. A final computation is performed to obtain shared $[\![T]\!]$ modulo 2^n of the result bit $t = t_0 \oplus t_1 \in \{0, 1\}$ shared modulo 2.

From the privacy point of view, when the seed s is (truly) random, $G(s)$ also looks like a random bit-string (this is a pseudorandom bit-string). Each half is used either in the cw or in the next state, but not both. Therefore, the correction words $CW^{(i)}$ do not contain information about the expected states and for $j = 0, 1$, the output k_j is independently uniformly distributed with respect to α and

Input: $(j, \mathsf{k}_j, [\![y]\!]_j)$ where $j \in \{0, 1\}$ refers to the evaluator id

Parse k_j as $[\![\alpha]\!]_j \,||\, s^{(1)} \,||\, CW^{(1)} \,||\, \cdots \,||\, CW^{(n+1)}$

Publish $[\![\alpha]\!]_j + [\![y]\!]_j \bmod 2^n$ and get revealed $x = \alpha + y \bmod 2^n$

Let $t^{(1)} \leftarrow j$

for $i = 1..n$ **do**

  $state \leftarrow G(s^{(i)}) \oplus (t^{(i)} \cdot CW^{(i)}) = (state_0, state_1)$

  Parse $s^{(i+1)} \,||\, t^{(i+1)} = state_{x[i]}$

return $[\![T]\!]_j \leftarrow (-1)^j \cdot \left(t^{(n+1)} \cdot CW^{(n+1)} + s^{(n+1)}\right) \bmod 2^n$

Algorithm 2: Eval: evaluation of the function key for zero test $y = 0$

$s^{(1)}_{1-j}$, in a computational way. As a consequence, at the end of the evaluation, for $j = 0, 1$, T_j also follows a distribution independent of α. Until the shared values are reconstructed, even a malicious adversary cannot learn anything about α nor the inputs of the other player.

Function keys should be sent to the evaluators in advance, which requires one extra communication of the size of the keys. We use the trick of [10] to reduce the size of each correction word in the keys, from $2(1 + \lambda)$ to $(2 + \lambda)$ by reusing the pseudo-random λ-bit string dedicated to the state used when leaving the special path for the state used for staying onto it, since for the latter state the only constraint is the pseudo-randomness of the bitstring.

3.2 Comparison

Our major contribution to the function secret sharing scheme is regarding comparison (which allows to tackle non-polynomial activation functions for neural networks): we build on the idea of the equality test to provide a synthetic and efficient protocol whose structure is very close from the previous one.

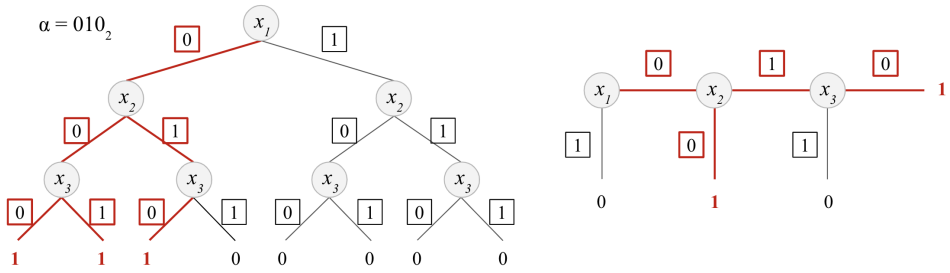

Figure 2: *(Left)* Binary decision tree with all the paths corresponding to $x \leq \alpha$ for n=3. *(Right)* Flat representation of the tree.

Instead of seeing the special path as a simple path, it can be seen as a frontier for the zone in the tree where $x \leq \alpha$. To evaluate $x \leq \alpha$, we could evaluate all the paths on the left of the special path and then sum up the results, but this is highly inefficient as it requires exponentially many evaluations. Our key idea here is to evaluate all these paths at the same time, noting that each time one leaves the special path, it either falls on the left side (i.e., $x < \alpha$) or on the right side (i.e., $x > \alpha$). Hence, we only need to add an extra step at each node of the evaluation, where depending on the bit value $x[i]$, we output a leaf label which is 1 only if $x[i] < \alpha[i]$ and all previous bits are identical. Only one label between the final label (which corresponds to $x = \alpha$) and the leaf labels can be equal to one, because only a single path can be taken. Therefore, evaluators will return the sum of all the labels to get the final output.

The full description of the comparison protocol is detailed in Appendix A, together with a detailed explanation of how it works.

4 Application to Deep Learning

We now apply these primitives to a private deep learning setup in which a model owner interacts with a data owner. The data and the model parameters are sensitive and are secret shared to be kept private. The shape of the input and the architecture of the model are however public, which is a standard assumption in secure deep learning [34, 36].

4.1 Additive sharing workflow

All our operations are modular and follow this additive sharing workflow: inputs are provided secret shared and are masked with random values before being revealed. This disclosed value is then consumed with preprocessed function keys to produce a secret shared output. Each operation is independent of all surrounding operations, which is known as *circuit-independent preprocessing* [11] and implies that key generation can be fully outsourced without having to know the model architecture. This results in a fast runtime execution with a very efficient online communication, with a single

round of communication and a message size equal to the input size for comparison. Preprocessing is performed by a trusted third party to build the function keys. This is a valid assumption in our use case as such third party would typically be an institution concerned about its image, and it is very easy to check that preprocessed material is correct using a *cut-and-choose* technique [51].

4.2 Machine Learning operations

Matrix Multiplication (MatMul). As mentioned by [11], multiplication fit in this additive sharing workflow. We use Beaver triples [2] to compute $[\![z]\!] = [\![x \cdot y]\!]$ from $[\![x]\!]$, $[\![y]\!]$ and using a triple $([\![a]\!], [\![b]\!], [\![c]\!] := [\![a \cdot b]\!])$, where all values are secret shared in $\mathbb{Z}_{2^n}$. The mask is here $[\![(-a, -b)]\!]$ and is used to reveal $(\delta, \epsilon) := (x - a, y - b)$. The functional keys are the shares of $([\![a]\!], [\![b]\!], [\![c]\!])$ and are used to compute $\delta \cdot [\![b]\!] + \epsilon \cdot [\![a]\!] + \delta \cdot \epsilon + [\![c]\!] = [\![z]\!]$. Matrix multiplication is identical but uses matrix Beaver triples [14].

ReLU activation function is supported as a direct application of our comparison protocol, which we combine with a point wise multiplication.

Convolution can be computed as a single matrix multiplication using an unrolling technique as described in [12] and illustrated in Figure 3 in Appendix C.2.

Argmax operator used in classification to determine the predicted label can also be computed in a constant number of rounds using pairwise comparisons as shown by [21]. The main idea here is, given a vector $(x_0, \ldots, x_{m-1})$, to compute the matrix $M \in \mathbb{R}^{m-1 \times m}$ where each row $M_i = (x_{i+1 \mod m}, \ldots, x_{i+m+1 \mod m})$. Then, each element of column j is compared to x_j, which requires $m(m-1)$ parallel comparisons. A column j where all elements are lower than x_j indicates that j is a valid result for the argmax.

MaxPool can be implemented by combining these two methods: the matrix is first unrolled like in Figure 3 and the maximum of each row in then computed using parallel pairwise comparisons. More details and an optimization when the kernel size k equals 2 are given in Appendix C.3.

BatchNorm is implemented using a approximate division with Newton's method as in [48]: given an input $\vec{x} = (x_0, \ldots, x_{m-1})$ with mean μ and variance σ^2, we return $\gamma \cdot \hat{\theta} \cdot (\vec{x} - \mu) + \beta$. Variables γ and β are learnable parameters and $\hat{\theta}$ is the estimate inverse of $\sqrt{\sigma^2 + \epsilon}$ with $\epsilon \ll 1$ and is computed iteratively using: $\theta_{i+1} = \theta_i \cdot (3 - (\sigma^2 + \epsilon) \cdot \theta_i^2)/2$. More details can be found in Appendix C.4.

More generally, for more complex activation functions such as softmax, we can use polynomial approximations methods, which achieve acceptable accuracy despite involving a higher number of rounds [37, 23, 21]. Table 1 summarizes the online communication cost of each operation, and shows that basic operations such as comparison have a very efficient online communication. We also report results from [48] which achieve good experimental performance.

4.3 Training phase using Autograd

These operations are sufficient to evaluate real world models in a fully private way. To also support private training of these models, we need to perform a private backward pass. As we overload operations such as convolutions or activation functions, we cannot use the built-in autograd functionality of PyTorch. Therefore, we have developed a custom autograd functionality, where we specify how to compute the derivatives of the operations that we have overloaded. Backpropagation also uses the same basic blocks than those used in the forward pass.

5 Extension to Private Federated Learning

This 2-party protocol between a model owner and a data owner can be extended to an n-party federated learning protocol where several *clients* contribute their data to a model owned by an orchestrator *server*. This approach is inspired by secure aggregation [8] but we do not consider here clients being phones which means we are less concerned with parties dropping before the end of the protocol. In addition, we do not reveal the updated model at each aggregation or at any stage, hence providing better privacy than secure aggregation.

At the beginning of the interaction, the server and model owner initializes its model and builds n pairs of additive shares of the model parameters. For each pair i, it keeps one of the shares and sends the

Table 1: Theoretical online communication complexity of our protocols. Input sizes into bracket are those of the layers' parameters, where k stands for the kernel size and s for the stride. Communication is given in number of values transmitted, and should be multiplied by their size (typically 4 bytes).

Protocol	Input size	Rounds		Communication	
		Ours	[48]	Ours	[48]
Equality	m	1	-	m	-
Comparison	m	1	7	m	$2m$
MatMul	$m_1 \times m_2, m_2 \times m_3$	1	1	$m_1m_2 + m_2m_3$	m_1m_3
Linear	$m_1 \times m_2, \{m_2 \times m_3\}$	1	1	$m_1m_2 + m_2m_3$	m_1m_3
Convolution	$m \times m, \{k, s\}$	1	1	$((m-k)/s+1)^2k^2 + k^2$	$\sim m^2k^2$
ReLU	m	2	10	$3m$	$4m$
Argmax	m	2	-	m^2	-
MaxPool	$m \times m, \{k, s\}$	3	$12(k^2-1)$	$((m-k)/s+1)^2(k^4+2)$	$\sim 5m^2$
BatchNorm	$m \times m$	9	335	$18m^2$	$\sim 56m^2$

Table 2: Inference time over several popular network architectures.

Model	Dataset	LAN Time (s)	WAN Time (s)	Online Comm. (MB)	Batch Size
Network-1	MNIST	0.004	0.055	0.013	128
Network-2	MNIST	0.096	0.234	0.33	128
LeNet	MNIST	0.129	0.289	0.46	128
AlexNet	CIFAR-10	0.303	0.696	1.46	128
AlexNet	64×64 ImageNet	0.723	1.36	2.56	128
VGG16	CIFAR-10	4.50	8.14	16.1	32
VGG16	64×64 ImageNet	11.4	35.9	102	16
ResNet18	224×224 ImageNet	43.3	80.8	220	4

other one to the corresponding client i. Then, the server runs in parallel the training procedure with all the clients until the aggregation phase starts. Aggregation for the server shares is straightforward, as the n shares it holds can be simply locally averaged. But the clients have to average their shares together to get a client share of the aggregated model. One possibility is that clients broadcast their shares and compute the average locally. However, to prevent a client colluding with the server from reconstructing the model contributed by a given client, they hide their shares using masking. This can be done using correlated random masks: client i generates a seed, sends it to client $i+1$ while receiving one from client $i-1$. Client i then generates a random mask M_i using its seed and another M_{i-1} using the one of client $i-1$ and publishes its share masked with $M_i - M_{i-1}$. As the masks cancel each other out, the computation will be correct.

6 Experiments

We follow a setup very close to [48] and assess inference and training performance of several networks on the datasets MNIST [33], CIFAR-10 [30], 64×64 Tiny Imagenet and 224×224 Tiny ImageNet [49, 42], presented in Appendix D.1. More precisely, we assess 5 networks as in [48]: a fully-connected network (Network-1), a small convolutional network with maxpool (Network-2), LeNet [32], AlexNet [31] and VGG16 [46]. Furthermore, we also include ResNet18 [22] which to the best of our knowledge has never been studied before in private deep learning. The description of these networks is taken verbatim from [48] and is available in Appendix D.2.

Our implementation is written in Python. To use our protocols that only work in finite groups like $\mathbb{Z}_{2^{32}}$, we convert our input values and model parameters to fixed precision. To do so, we rely on the PySyft library [43] which extends common deep learning frameworks including PyTorch with a communication layer for federated learning and supports fixed precision. The experiments are run on Amazon EC2 m5d.4xlarge machines with 16 cores and 64GB of RAM, and we report our results both in the LAN and in the WAN setting. Latency is of 70ms for the WAN setting and is considered negligible in the LAN setting. Last, all values are encoded on 32 bits.

Evaluation inference time. Comparison of experimental runtimes should be taken with caution, as different implementations and hardware may result in significant differences even for the same

Table 3: Accuracy of network architectures trained in plain text, tested in the private setting.

Model	Dataset	Private Accuracy	Normal Accuracy
Network-1	MNIST	98.1	98.1
Network-2	MNIST	99.0	99.0
LeNet	MNIST	99.4	99.4
AlexNet	CIFAR-10	60.6	60.7
AlexNet	64×64 Tiny ImageNet	33.3	33.7
VGG16	CIFAR-10	88.8	88.8
VGG16	64×64 Tiny ImageNet	41.1	41.3

Table 4: Training time and accuracy for networks privately trained from scratch (except AlexNet).

Model	Dataset	LAN Time (h)	Accuracy	Nb of epochs
Network-1	MNIST	5.46	97.8	15
Network-2	MNIST	82	87.3	12
LeNet	MNIST	51	96.6	6
AlexNet	CIFAR-10	58	62.3	4

protocol. However, our inference runtimes reported in Table 2 compare favourably with existing work including [34–36, 47, 48], in the LAN setting and particularly in the WAN setting thanks to our reduced number of communication rounds. For example, our implementation of Network-1 is $2\times$ faster than the best previous result by [35] in the LAN setting and $18\times$ faster in the WAN setting compared to [48]. For bigger networks such as AlexNet on CIFAR-10, we are still $13\times$ faster in the WAN setting than [48]. Results are given for a batched evaluation, which allows parallelism and hence faster execution as in [48]. For larger networks, we reduce the batch size to have the preprocessing material (including the function keys) fitting into RAM.

Test accuracy. Thanks to the flexibility of our framework, we can train each of these networks in plain text and need only one line of code to turn them into private networks, where all parameters are secret shared. We compare these private networks to their plaintext counterparts and observe that the accuracy is well preserved as shown in Table 3. If we degrade the encoding precision, which by default considers values in $\mathbb{Z}_{2^{32}}$, and the fixed precision which is by default of 3 decimals, performance degrades as shown in Appendix B.

Training. We can either train from scratch those networks or fine tune pre-trained models. Training is an end-to-end private procedure, which means the loss and the gradients are never accessible in plain text. We use stochastic gradient descent (SGD) which is a simple but popular optimizer, and support both hinge loss and mean square error (MSE) loss, as other losses like cross entropy which is used in clear text by [48] cannot be computed over secret shared data without approximations. We report runtime and accuracy obtained by training from scratch the smaller networks in Table 4. Note that because of the number of epochs, the optimizer and the loss chosen, accuracy does not match best known results. However, the training procedure is not altered and the trained model will be strictly equivalent to its plaintext counterpart. Training cannot complete in reasonable time for larger networks, which are anyway available pre-trained. Note that training time includes the time spent building the preprocessing material, as it cannot be fully processed in advance and stored in RAM.

Discussion. For larger networks, we could not use batches of size 128. This is mainly due to the size of the comparison function keys which is currently proportional to the size of the input tensor, with a multiplication factor of $n\lambda$ where $n = 32$ and $\lambda = 128$. Optimizing the function secret sharing protocol to reduce those keys would lead to massive improvements in the protocol's efficiency.

Our implementation actually has more communication than is theoretically necessary according to Table 1, suggesting that the experimental results could be further improved. As we build on top of PyTorch, using machines with GPUs could also potentially result in a massive speed-up, as an important fraction of the execution time is dedicated to computation.

Last, accuracies presented in Table 3 and Table 4 do not match state-of-the-art performance for the models and datasets considered. This is not due to internal defaults of our protocol but to the simplified training procedure we had to use. Supporting losses such as the logistic loss, more complex optimizers like Adam and dropout layers would be an interesting follow-up.

Acknowledgments

We would like to thank Geoffroy Couteau, Chloé Hébant and Loïc Estève for helpful discussions throughout this project. We are also grateful for the long-standing support of the OpenMined community and in particular its dedicated cryptography team, including Yugandhar Tripathi, S P Sharan, George-Cristian Muraru, Muhammed Abogazia, Alan Aboudib, Ayoub Benaissa, Sukhad Joshi and many others.

This work was supported in part by the European Community's Seventh Framework Programme (FP7/2007-2013 Grant Agreement no. 339563 – CryptoCloud) and by the French project FUI ANBLIC. The computing power was graciously provided by the French company ARKHN.

References

[1] Mohammad Al-Rubaie and J. Morris Chang. Privacy-preserving machine learning: Threats and solutions. *IEEE Security & Privacy*, 17(2):49–58, 2019.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[3] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the Conference on Computer and Communications Security*, pages 578–590, 2016.

[4] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 45–56, 2019.

[5] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 3–13, 2019.

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[7] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, and H. Brendan McMahan. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[8] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 337–367. Springer, 2015.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[11] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2019.

[12] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *international Conference on the Theory and Application of Cryptology and Information Security*, pages 3–33. Springer, 2016.

[14] Morten Dahl. Private Image Analysis with MPC. *Accessed 2019-11-01. Available: https://mortendahl.github.io/2017/09/19/private-image-analysis-with-mpc/*, 2017.

[15] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[17] Tamara Dugan and Xukai Zou. A survey of secure multiparty computation protocols for privacy preserving genetic tests. In *International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 173–182. IEEE, 2016.

[18] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the Conference on Computer and Communications Security*, pages 1322–1333, 2015.

[19] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[20] Oded Goldreich. *Foundations of Cryptography: volume 2, Basic Applications*. Cambridge University Press, 2009.

[21] Awni Hannun, Brian Knott, Shubho Sengupta, and Laurens van der Maaten. Privacy-preserving contextual bandits. *arXiv preprint arXiv:1910.05299*, 2019.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[23] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*, volume 80. Siam, 2002.

[24] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 603–618, 2017.

[25] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

[26] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *USENIX Security Symposium 18*, pages 1651–1669, 2018.

[27] Harmanjeet Kaur, Neeraj Kumar, and Shalini Batra. An efficient multi-party scheme for privacy preserving collaborative filtering for healthcare recommender system. *Future Generation Computer Systems*, 86:297–307, 2018.

[28] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[29] Jakub Konečnỳ, H Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[30] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55, 2014.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[32] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[33] Yann LeCun, Corinna Cortes, and C. J. Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[34] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the Conference on Computer and Communications Security*, pages 619–631, 2017.

[35] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the Conference on Computer and Communications Security*, pages 35–52, 2018.

[36] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

[37] Victor Pan and Robert Schreiber. An improved newton iteration for the generalized inverse of a matrix, with applications. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1109–1130, 1991.

[38] Sangchul Park, Gina Jeehyun Choi, and Haksoo Ko. Information technology–based tracing strategy in response to COVID-19 in South Korea—privacy controversies. *JAMA*, 2020.

[39] Leonie Reichert, Samuel Brack, and Björn Scheuermann. Privacy-preserving contact tracing of covid-19 patients. *Cryptology ePrint*, (2020/375), 2020.

[40] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the Asia Conference on Computer and Communications Security*, pages 707–721, 2018.

[41] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, and Michael Bernstein. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[43] Théo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. A generic framework for privacy preserving deep learning. *arXiv preprint arXiv:1811.04017*, 2018.

[44] Microsoft SEAL (release 3.0). `http://sealcrypto.org`, October 2018. Microsoft Research, Redmond, WA.

[45] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the Conference on Computer and Communications Security*, pages 1310–1321, 2015.

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[47] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: Efficient and private neural network training. *IACR Cryptology ePrint Archive*, 2018:442, 2018.

[48] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *arXiv preprint arXiv:2004.02229*, 2020.

[49] Jiayu Wu, Qixiang Zhang, and Guoxi Xu. Tiny imagenet challenge. Technical report, Available: `http://cs231n.stanford.edu/reports/2017/pdfs/930.pdf`, 2017.

[50] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Annual Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[51] Ruiyu Zhu, Yan Huang, Jonathan Katz, and Abhi Shelat. The cut-and-choose game and its application to cryptographic protocols. In *USENIX Security Symposium Security 16*, pages 1085–1100, 2016.

A Function Secret Sharing - Comparison

One can observe the great similarity of structure of the comparison protocol given in Algorithm 3 and 4 with the equality protocol from Algorithm 1 and 2: the equality test is performed in parallel with an additional information out_i at each node, which holds a share of either 0 when the evaluator stays on the special path or if it has already left it at a previous node, or a share of $\alpha[i]$ when it leaves the special path. This means that if $\alpha[i] = 1$, leaving the special path implies that $x[i] = 0$ and hence $x \leq \alpha$, while if $\alpha[i] = 0$, leaving implies $x[i] = 1$ so $x > \alpha$ and the output should be 0. The final share out_{n+1} corresponds the previous equality test.

Note that in all these computations modulo 2^n, while the bitstrings $s_j^{(i)}$ and $\sigma_j^{(i)}$ are in $\{0,1\}^\lambda$, they are always either the same for the two players or compensated by the $CW^{(i)}$ and $CW_{leaf}^{(i)}$ respectively, hence they cancel. In the end, the sum $[\![T]\!]_j$ of the out_i's is a share of 1 either if out_{n+1} was a share of 1 (i.e. $x = \alpha$) or if one of the other out_i was, which is possible only if $\alpha[i] = 1$ and $x[i] < \alpha[i]$ (i.e. $x < \alpha$). Otherwise, $[\![T_j]\!]$ is a share of 0 and $x > \alpha$.

Let $G : \{0,1\}^\lambda \to \{0,1\}^{4(\lambda+1)}$ be a pseudorandom generator, where the output is seen as a pair of pairs of elements in $\{0,1\}^{\lambda+1}$, and so $(\{0,1\}^{\lambda+1} \times \{0,1\}^{\lambda+1}) \times (\{0,1\}^{\lambda+1} \times \{0,1\}^{\lambda+1})$.

Initialisation: Sample random $\alpha \xleftarrow{\$} \mathbb{Z}_{2^n}$
Sample random $s_j^{(1)} \xleftarrow{\$} \{0,1\}^\lambda$ and set $t_j^{(1)} \leftarrow j$, for $j = 0, 1$

for $i = 1..n$ **do**
  for $j = 0, 1$ **do**
    $((s_j^{\mathsf{L}} \,||\, t_j^{\mathsf{L}},\ ||\ s_j^{\mathsf{R}} \,||\, t_j^{\mathsf{R}}), (\sigma_j^{\mathsf{L}} \,||\, \tau_j^{\mathsf{L}}, \sigma_j^{\mathsf{R}} \,||\, \tau_j^{\mathsf{R}})) \leftarrow G(s_j^{(i)})$
  if $\alpha[i]$ **then** $cw^{(i)} \leftarrow \big((0^\lambda \,||\, 0, s_0^{\mathsf{L}} \oplus s_1^{\mathsf{L}} \,||\, 1), (\sigma_0^{\mathsf{R}} \oplus \sigma_1^{\mathsf{R}} \,||\, 1, 0^\lambda \,||\, 0)\big)$
  else $cw^{(i)} \leftarrow \big((s_0^{\mathsf{R}} \oplus s_1^{\mathsf{R}} \,||\, 1, 0^\lambda \,||\, 0), (0^\lambda \,||\, 0, \sigma_0^{\mathsf{L}} \oplus \sigma_1^{\mathsf{L}} \,||\, 1)\big)$;
  $CW^{(i)} \leftarrow cw^{(i)} \oplus G(s_0^{(i)}) \oplus G(s_1^{(i)})$
  for $j = 0, 1$ **do**
    $state_j \leftarrow G(s_j^{(i)}) \oplus (t_j^{(i)} \cdot CW^{(i)}) = ((state_{j,0}, state_{j,1}), (state'_{j,0}, state'_{j,1}))$
    Parse $s_j^{(i+1)} \,||\, t_j^{(i+1)} = state_{j,\alpha[i]}$ and $\sigma_j^{(i+1)} \,||\, \tau_j^{(i+1)} = state'_{j,1-\alpha[i]}$
  $CW_{leaf}^{(i)} \leftarrow (-1)^{\tau_1^{(i+1)}} \cdot [\sigma_1^{(i+1)} - \sigma_0^{(i+1)} + \alpha[i]] \bmod 2^n$
$CW_{leaf}^{(n+1)} \leftarrow (-1)^{t_1^{(n+1)}} \cdot \left(1 - s_0^{(n+1)} + s_1^{(n+1)}\right) \bmod 2^n$
return $\mathsf{k_j} \leftarrow [\![\alpha]\!]_j \,||\, s_j^{(1)} \,||\, (CW^{(i)})_{i=1..n} \,||\, (CW_{leaf}^{(i)})_{i=1..n+1}$, for $j = 0, 1$

Algorithm 3: KeyGen: key generation for comparison to α

Input: $(j, \mathsf{k}_j, [\![y]\!]_j)$ where $j \in \{0,1\}$ refers to the evaluator id
Parse k_j as $[\![\alpha]\!]_j \,||\, s^{(1)} \,||\, (CW^{(i)})_{i=1..n} \,||\, (CW_{leaf}^{(i)})_{i=1..n+1}$
Publish $[\![\alpha]\!]_j + [\![y]\!]_j$ and get revealed $x = \alpha + y \bmod 2^n$
Let $t^{(1)} \leftarrow j$
for $i = 1..n$ **do**
  $state \leftarrow G(s^{(i)}) \oplus (t^{(i)} \cdot CW^{(i)}) = (state_0, state_1)$
  Parse $(s^{(i+1)} \,||\, t^{(i+1)}, \sigma_j^{(i+1)} \,||\, \tau_j^{(i+1)}) = state_{x[i]}$
  $out_i \leftarrow (-1)^j \cdot \left(\tau^{(i+1)} \cdot CW_{leaf}^{(i)} + \sigma^{(i+1)}\right) \bmod 2^n$
$out_{n+1} \leftarrow (-1)^j \cdot \left(t^{(n+1)} \cdot CW_{leaf}^{(n+1)} + s^{(n+1)}\right) \bmod 2^n$
return $[\![T]\!]_j \leftarrow \sum_i out_i \bmod 2^n$

Algorithm 4: Eval: evaluation of the function key for comparison $y \leq 0$

B Encoding precision

We have studied the impact of lowering the encoding space of the input to our function secret sharing protocol from $\mathbb{Z}_{2^{32}}$ to $\mathbb{Z}_{2^k}$ with $k < 32$. Finding the lowest k guaranteeing good performance is an interesting challenge as the function keys size is directly proportional to it. This has to be done together with reducing fixed precision from 3 decimals down to 1 decimal to ensure private values aren't too big, which would result in higher failure rate in our private comparison protocol. We have reported in Table 5 our findings on Network-1, which is pre-trained and then evaluated in a private fashion.

Decimals	$\mathbb{Z}_{2^{12}}$	$\mathbb{Z}_{2^{16}}$	$\mathbb{Z}_{2^{20}}$	$\mathbb{Z}_{2^{24}}$	$\mathbb{Z}_{2^{28}}$	$\mathbb{Z}_{2^{32}}$
1	-	-	-	-	-	9.5
2	69.4	96.0	97.9	98.1	98.0	98.1
3	10.4	76.2	96.9	98.1	98.2	98.1
4	9.7	14.3	83.5	97.4	98.1	98.2

Table 5: Accuracy (in %) of Network-1 given different precision and encoding spaces

What we observe is that 3 decimals of precision is the most appropriate setting to have an optimal precision while allowing to slightly reduce the encoding space down to $\mathbb{Z}_{2^{24}}$ or $\mathbb{Z}_{2^{28}}$. Because this is not a massive gain and in order to keep the failure rate in comparison very low, we have kept $\mathbb{Z}_{2^{32}}$ for all our experiments.

C Implementation details

C.1 Failure rate of FSS comparison in practical scenarios

Our comparison protocol can fail if $y + \alpha$ wraps around and becomes negative. We can't act on α because it must be completely random to act as a perfect mask and to make sure the revealed $x = y + \alpha \bmod 2^n$ does not leak any information about y, but the smaller y is, the lower the error probability will be. [11] suggests a method which uses 2 invocations of the protocol to guarantee perfect correctness but because it incurs an important runtime overhead, we rather show that the failure rate of our comparison protocol is very small and is reasonable in contexts that tolerate a few mistakes, as in machine learning. More precisely, we quantify it on real world examples, namely on Network-2 and on the 64×64 Tiny Imagenet version of VGG16, with a fixed precision of 3 decimals, and find respective failure rates of 1 in 4 millions comparisons and 1 in 100 millions comparisons. Such error rates do not affect the model accuracy, as Table 3 shows.

C.2 Unrolling convolutions

Figure 3 illustrates how to transform a convolution operation into a single matrix multiplication.

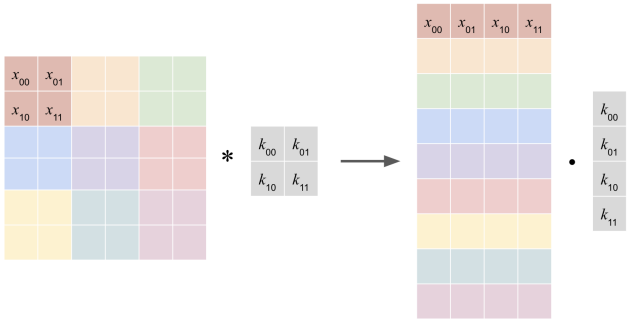

Figure 3: Illustration of unrolling a convolution with kernel size $k = 2$ and stride $s = 2$.

C.3 MaxPool and optimisation

Figure 4 illustrates how MaxPool uses ideas from matrix unrolling and argmax computation. Notations present in the figure are consistent with the explanation of argmax using pairwise comparison in Section 4.3. The $m \times m$ matrix is first unrolled to a $m^2 \times k^2$ matrix. It is then expanded on k^2 layers, each of which each shifted by a step of 1. Next, $m^2k^2(k^2-1)$ pairwise comparisons are then applied simultaneously between the first layer and the other ones, and for each x_i we sum the result of its $k-1$ comparison and check if it equals $k-1$. We multiply this boolean by x_i and sum up along a line (like x_1 to x_4 in the figure). Last, we restructure the matrix back to its initial structure.

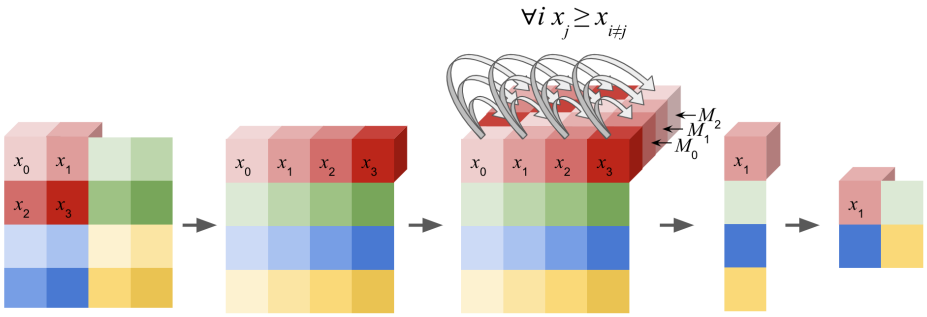

Figure 4: Illustration of MaxPool with kernel size $k = 2$ and stride $s = 2$.

In addition, when the kernel size k is 2, rows are only of length 4 and it can be more efficient to use a binary tree approach instead, i.e. compute the maximum of columns 0 and 1, 2 and 3 and the max of the result: it requires $\log_2(k^2) = 2$ rounds of communication and only approximately $(k^2-1)(m/s)^2$ comparisons, compared to a fixed 3 rounds and approximately $k^4(m/s)^2$.

Interestingly, average pooling can be computed locally on the shares without interaction because it only includes mean operations, but we didn't replace MaxPool operations with average pooling to avoid distorting existing neural networks architecture.

C.4 BatchNorm approximation

The BatchNorm layer is the only one in our implementation which is a polynomial approximation. Moreover, compared to [48], the approximation is significantly coarser as we don't make any costly initial approximation and we reduce the number of iterations of the Newton method from 4 to only 3. Typical relative error can be up to 20% but as the primary purpose of BatchNorm is to normalise data, having rough approximations here is not an issue and doesn't affect learning capabilities, as our experiments show. However, it is a limitation for using pre-trained networks: we observed on AlexNet adapted to CIFAR-10 that training the model with a standard BatchNorm and evaluating it with our approximation resulted in poor results, so we had to train it with the approximated layer.

D Datasets and Networks Architecture

D.1 Datasets

This section is taken almost verbatim from [48].

We select 4 datasets popularly used for training image classification models: MNIST [33], CIFAR-10 [30], 64×64 Tiny Imagenet and 224×224 Tiny ImageNet [49].

MNIST MNIST [33] is a collection of handwritten digits dataset. It consists of 60,000 images in the training set and 10,000 in the test set. Each image is a 28×28 pixel image of a handwritten digit along wit a label between 0 and 9. We evaluate Network-1, Network-2, and the LeNet network on this dataset.

CIFAR-10 CIFAR-10 [30] consists of 50,000 images in the training set and 10,000 in the test set. It is composed of 10 different classes (such as airplanes, dogs, horses etc.) and there are 6,000 images

of each class with each image consisting of a colored 32×32 image. We perform private training of AlexNet and inference of VGG16 on this dataset.

Tiny ImageNet Tiny ImageNet [49] consists of two datasets of 100,000 training samples and 10,000 test samples with 200 different classes. The first dataset is composed of colored 64×64 images and we use it with AlexNet and VGG16. The second is composed of colored 224×224 images and is used with ResNet18.

D.2 Model description

We have selected 6 models for our experimentations.

Network-1 A 3-layered fully-connected network with ReLU used in SecureML [36].

Network-2 A 4-layered network selected in MiniONN [34] with 2 convolutional and 2 fully-connected layers, which uses MaxPool in addition to ReLU activation.

LeNet This network, first proposed by LeCun et al. [32], was used in automated detection of zip codes and digit recognition. The network contains 2 convolutional layers and 2 fully connected layers.

AlexNet AlexNet is the famous winner of the 2012 ImageNet ILSVRC-2012 competition [31]. It has 5 convolutional layers and 3 fully connected layers and it can batch normalization layer for stability and efficient training.

VGG16 VGG16 is the runner-up of the ILSVRC-2014 competition [46]. VGG16 has 16 layers and has about 138M parameters.

ResNet18 ResNet18 [22] is the runner-up of the ILSVRC-2015 competition. It is a convolutional neural network that is 18 layers deep, and has 11.7M parameters. It uses batch normalisation and we're the first private deep learning framework to evaluate this network.

D.3 Model Architecture

Model architectures of Network-1 and Network-2, together with LeNet, and the adaptations for CIFAR-10 of AlexNet and VGG16 are precisely depicted in Appendix D of [48]. Note that in the CIFAR-10 version AlexNet, authors have used the version with BatchNorm layers, and we have kept this choice. For the 64×64 Tiny Imagenet version of AlexNet, we used the standard architecture from PyTorch to have a pretrained network. It doesn't have BatchNorm layers, and we have adapted the classifier part as illustrated in Figure 5. Note also that we permute ReLU and Maxpool where applicable like in [48], as this is strictly equivalent in terms of output for the network and reduces the number of comparisons. More generally, we don't proceed to any alteration of the network behaviour except with the approximation on BatchNorm. This improves usability of our framework as it allows to take a pre-trained neural network from a standard deep learning library like PyTorch and to encrypt it generically with a single line of code.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU()
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU()
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1)
  )
  (classifier): Sequential(
    (0): Linear(in_features=256, out_features=4096)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=4096, out_features=4096)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=4096, out_features=200)
  )
)
```

Figure 5: Adaptation of AlexNet to the Tiny Imagenet dataset